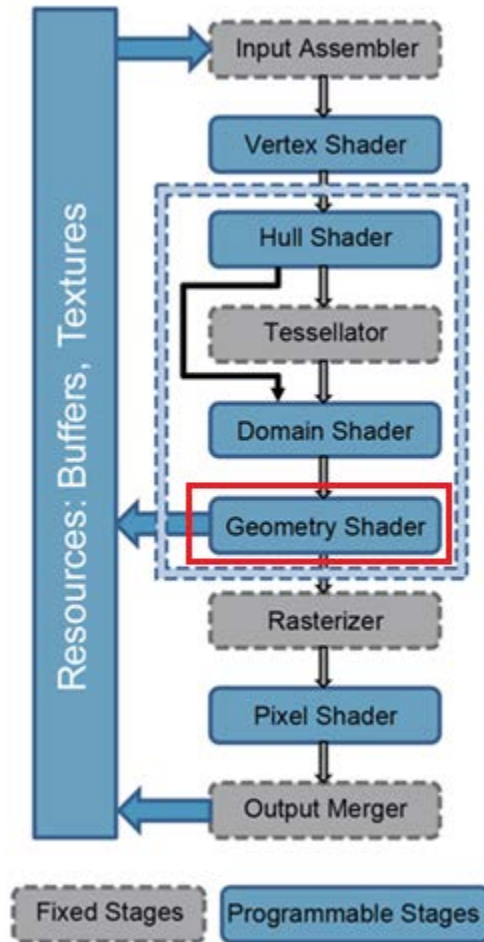


Geometry Shader

Geometry Shader

- The Geometry Shader is an optional stage that sits between vertex shader and fragment shader.
- The vertex shader input vertices the geometry shader inputs geometries (points, triangles, polygons)
- The main advantage of GS is they can create or destroy complete geometries

Geometry Shader



Geometry Shaders

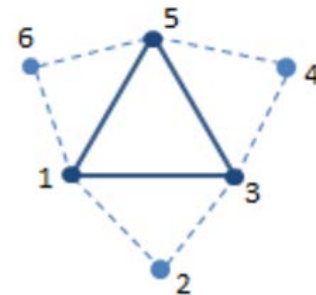
- So one primitive can be expanded into other primitive types.
- Points can be converted to triangles.
- And triangles into many triangles.

Primitive Types

- The available geometry shader input primitives are:
 - points (1)
 - lines (2)
 - triangles (3)
 - lines_adjacency (4)
 - triangles_adjacency (6)

Primitive Types

- while primitives without adjacency contain only the vertices of the target primitive.
- Primitives with adjacency contain some of the surrounding vertices, Lines Adjacency, Triangles Adjacency



Data Types

- `gl_InvocationID` — contains the invocation index of the current shader.
- It is assigned an integer value in the range $[0, N-1]$ where N is the number of geometry shader invocations per primitive
- Identifies the invocation number assigned to the geometry shader invocation.

Built In Data Types

- Built in struct
in `gl_PerVertex` {
 `vec4 gl_Position;`
 `float gl_PointSize;`
 `float gl_ClipDistance[];`
} `gl_in[];`
- Get information on a per vertex basis.
- Gets an **array** of information.

Built In Data Types

- `gl_Position` information is transferred from one shader stage to the next.
- For Example
 - Vertex Shader
 - `layout (location = 0) in vec3 position;`
 - `gl_Position = vec4(position, 1.0f);`
 - Geometry Shader
 - `vec4 p1 = gl_in[0].gl_Position`
- Then `p1` will have same value as what was assigned into `gl_Position` in vertex shader.
- In vertex shader if u multiply by `mvp` then `p1` will be affected accordingly

Custom Data Types

```
out VS_GS_VERTEX{
    out vec4 position;
    out vec3 color;
    out mat4 mvp;
} vs_out;

void main(){

    gl_Position = mvp * vec4(position, 1.0f);

    vs_out.color = color;
    vs_out.position = gl_Position;
    vs_out.mvp = mvp;
}
```

Custom Data Types

```
in VS_GS_VERTEX{  
    in vec4 position;  
    in vec3 color;  
    in mat4 mvp;  
}gs_in[];  
out vec3 outColor;  
void main() {  
    outColor = gs_in[0].color;  
    gl_Position = gs_in[0].position + gs_in[0].mvp * vec4(-2.0f,  
0.0f, 0.0f, 0.0f);  
}
```

GS Special Properties - Instancing

- You can cause the GS to execute multiple times for the same input primitive.
- Each invocation of the GS for a particular input primitive will get a different **gl_InvocationID** value.
- This is useful for layered rendering and outputs to multiple streams.

layout(invocations = num_instances) in;

GS Special Properties - Output Streams

- GS's can send vertex data to one stream to fragment shader.
- While building per-instance data in another stream.
- Multiple stream output *requires* that the output primitive type be points.
- You can still take whatever input you prefer.

GS Special Properties - Output Streams

- output variables can be given a stream index with a layout qualifier
- `layout(stream = stream_index) out vec4 some_output;`
- The *stream_index* ranges from 0 to `GL_MAX_VERTEX_STREAMS - 1`
- A default value for the stream can be set
- `layout(stream = 2) out;`

Geometry Shaders

- The shader itself is a bit different from the vertex and fragment shader.
- The type of data being passed in must be specified in the geometry shader layout (points) in;
- Here we are passing in points but we could be passing in lines or triangles as well.

Geometry Shaders

- We also need to specify the type of primitive we want out of the shader.
- Also the maximum number of vertices being sent out should also be specified.

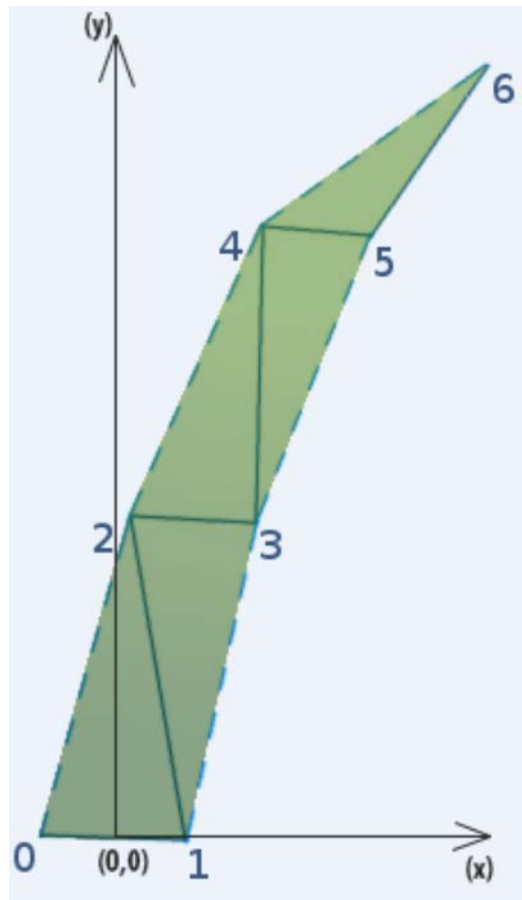
```
layout (triangle_strip, max_vertices = 3)  
out;
```

- `out` is a keyword which specifies the output type.
- Here we can be sending out *points* or *line_strip* instead.
- The max vertices being sent out is 3

Geometry Shaders

- The new vertices are specified with respect to the objects local coordinates.
- `gl_Position = gs_in[0].position + vec4(-2.0f, 0.0f, 0.0f, 0.0f);`

Geometry Shaders



Geometry Shaders

- Every time a vertex is added **EmitVertex()** needs to be called as the vertex will be added to the primitive.
- Once the number of vertices is added to form the shape **EndPrimitive()** needs to be called to create the primitives from vertices added.

```
#version 430 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
```

```
out VS_GS_VERTEX{
    out vec4 position;
    out vec3 color;
    out mat4 mvp;
} vs_out;
```

```
uniform mat4 mvp;
```

```
void main(){
```

```
    gl_Position = mvp * vec4(position, 1.0f);
```

```
    vs_out.color = color;
```

```
    vs_out.position = gl_Position;
```

```
    vs_out.mvp = mvp;
```

```
}
```

Vertex Shader



```
#version 430 core
layout (points) in;
layout (triangle_strip, max_vertices = 3) out;
```

```
out vec3 outColor;
```

```
in VS_GS_VERTEX{
    in vec4 position;
    in vec3 color;
    in mat4 mvp;
}gs_in[];
```

Geometry Shader

```
void main() {
```

```
    outColor = gs_in[0].color;
```

```
    gl_Position = gs_in[0].position + gs_in[0].mvp * vec4(-2.0f, 0.0f, 0.0f, 0.0f); EmitVertex();
```

```
    gl_Position = gs_in[0].position + gs_in[0].mvp * vec4(2.0f, 0.0f, 0.0f, 0.0f); EmitVertex();
```

```
    gl_Position = gs_in[0].position + gs_in[0].mvp * vec4(0.0f, 2.0f, 0.0f, 0.0f); EmitVertex();
```

```
    EndPrimitive();
```

```
}
```



Fragement Shader

```
#version 430 core
```

```
in vec3 outColor;  
out vec4 color;
```

```
void main(){
```

```
    color = vec4(outColor, 1.0f);
```

```
}
```

```
GeometryModel::GeometryModel(GLuint program, Camera* camera){
```

```
    this->program = program;
```

```
    this->camera = camera;
```

```
    GLfloat points[] = {  
        0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, //passing in 1 point  
    };
```

```
    glBindVertexArray(vao);  
    glGenBuffers(1, &vbo);
```

Geometry Model.cpp

```
    glGenVertexArrays(1, &vao);  
    glBindBuffer(GL_ARRAY_BUFFER, vbo);  
    glBufferData(GL_ARRAY_BUFFER, sizeof(points), &points, GL_STATIC_DRAW);
```

```
    glEnableVertexAttribArray(0);  
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), 0);  
    glEnableVertexAttribArray(1);  
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
```

```
    glBindVertexArray(0);  
}
```



Geometry Model Render Function

```
void GeometryModel ::render() {  
  
    glUseProgram(this->program);  
  
    glm::mat4 model;  
    model = glm::translate(model, position);  
  
    glm::mat4 mvp = camera->getprojectionMatrix() * camera->getViewMatrix() * model;  
  
    GLint vpLoc = glGetUniformLocation(program, "mvp");  
    glUniformMatrix4fv(vpLoc, 1, GL_FALSE, glm::value_ptr(mvp));  
  
    glBindVertexArray(vao);  
    glDrawArrays(GL_POINTS, 0, 1);  
    glBindVertexArray(0);  
  
}
```


Shader Loader

- Make changes to shaderLoader.h and .cpp to take in compile, attach and link vs, gs and fs.
- Add function to take all three shader in ShaderLoader.h

```
GLuint CreateProgram(char*  
    vertexShaderFilename, char*  
    fragmentShaderFilename, char*  
    geometryShaderFilename);
```

Shader Loader.cpp

- Copy and paste the createProgram function that takes only the vs and fs.
- In the new createProgram function edit the function to accept gs as well.
- Then add in the following.
- `std::string geometry_shader_code = ReadShader(geometryShaderFilename);`
- `GLuint geometry_shader = CreateShader(GL_GEOMETRY_SHADER, geometry_shader_code, "geometry shader");`
- `glAttachShader(program, geometry_shader);`



Usage

- In the main.cpp

```
#include "GeometryModel.h"  
GeometryModel *geomModel;
```

- In init function

```
GLuint geomProgram =  
    shader.CreateProgram("Assets/shaders/geomModel.vs",  
        "Assets/shaders/geomModel.fs",  
        "Assets/shaders/geomModel.gs");
```

```
geomModel = new GeometryModel(geomProgram, camera);  
geomModel->setPosition(glm::vec3(6.0f, 1.0f, 0.0f));
```

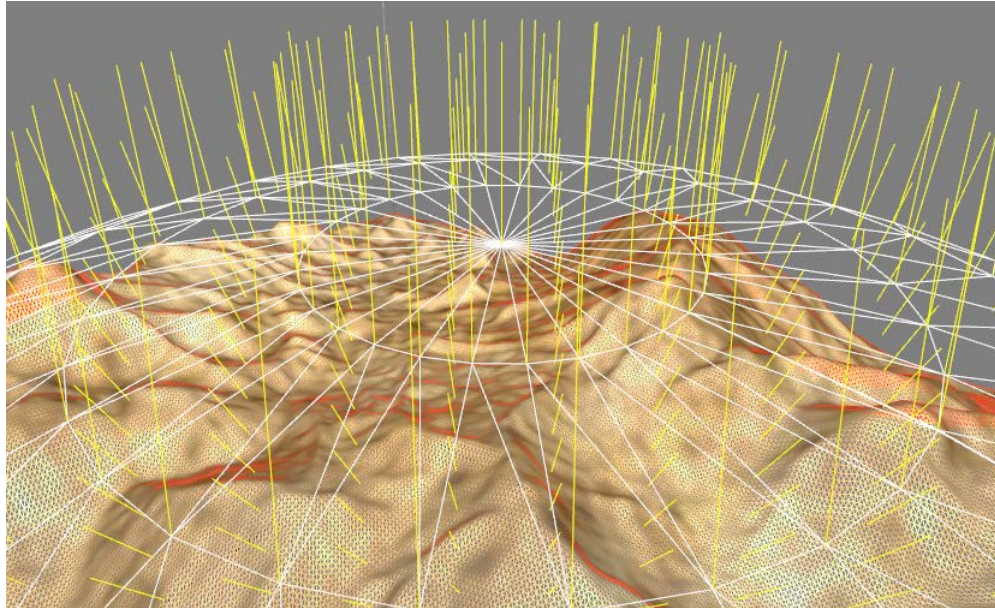
Usage

- In the render function in main.cpp render the geometry model

```
geomModel->render();
```

- The output should be a triangle though only a point is passed in

Applications



- Show geometry normals

Applications



- Reference
 - https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch07.html

Exercises

- Implement geometry shader.
 - Take a point as an input and creat a quad out of it.
 - Create texture coordinates for each new point created.
 - Apply a grass texture on the quad created.
 - Try rotating the quad locally
 - Move or rotate the quad in the world space.